PL/SQL

```
CREATE TABLE Customers (
    CustomerID NUMBER PRIMARY KEY,
    Name VARCHAR2(100),
    DOB DATE,
    Balance NUMBER,
    LastModified DATE
);

CREATE TABLE Accounts (
    AccountID NUMBER PRIMARY KEY,
    CustomerID NUMBER,
    AccountType VARCHAR2(20),
    Balance NUMBER,
    LastModified DATE,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);

CREATE TABLE Transactions (
    TransactionID NUMBER PRIMARY KEY,
    AccountID NUMBER,
    TransactionDate DATE,
    Amount NUMBER,
    TransactionType VARCHAR2(10),
    FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)
);

CREATE TABLE Loans (
```

```
    LoanID NUMBER PRIMARY KEY,
    CustomerID NUMBER,
    LoanAmount NUMBER,
    InterestRate NUMBER,
    StartDate DATE,
    EndDate DATE,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);

CREATE TABLE Employees (
    EmployeeID NUMBER PRIMARY KEY,
    Name VARCHAR2(100),
    Position VARCHAR2(50),
    Salary NUMBER,
    Department VARCHAR2(50),
    HireDate DATE
);
INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
VALUES (1, 'John Doe', TO_DATE('1985-05-15', 'YYYY-MM-DD'), 1000, SYSDATE);

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
VALUES (2, 'Jane Smith', TO_DATE('1990-07-20', 'YYYY-MM-DD'), 1500,
SYSDATE);

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance,
LastModified)
VALUES (1, 1, 'Savings', 1000, SYSDATE);

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance,
LastModified)
```

VALUES (2, 2, 'Checking', 1500, SYSDATE);

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (1, 1, SYSDATE, 200, 'Deposit');

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (2, 2, SYSDATE, 300, 'Withdrawal');

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate)

VALUES (1, 1, 5000, 5, SYSDATE, ADD_MONTHS(SYSDATE, 60));

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)

VALUES (1, 'Alice Johnson', 'Manager', 70000, 'HR', TO_DATE('2015-06-15', 'YYYY-MM-DD'));

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)

VALUES (2, 'Bob Brown', 'Developer', 60000, 'IT', TO_DATE('2017-03-20', 'YYYY-MM-DD'));

Exercise 1: Control Structures

Scenario 1: The bank wants to apply a discount to loan interest rates for customers above 60 years old.

- **Question:** Write a PL/SQL block that loops through all customers, checks their age, and if they are above 60, apply a 1% discount to their current loan interest rates.

Answer:

DECLARE

```
    CURSOR c_customers IS
        SELECT CustomerID, (FLOOR((SYSDATE - DOB) / 365)) AS Age
        FROM Customers;
    v_CustomerID Customers.CustomerID%TYPE;
    v_Age NUMBER;
BEGIN
    FOR customer_rec IN c_customers LOOP
        v_CustomerID := customer_rec.CustomerID;
        v_Age := customer_rec.Age;
        IF v_Age > 60 THEN
            UPDATE Loans
            SET InterestRate = InterestRate - 1
            WHERE CustomerID = v_CustomerID;
        END IF;
    END LOOP;
    COMMIT;
END;
```

Scenario 2: A customer can be promoted to VIP status based on their balance.

- **Question:** Write a PL/SQL block that iterates through all customers and sets a flag IsVIP to TRUE for those with a balance over $10,000.

```
ALTER TABLE Customers ADD IsVIP VARCHAR2(3) DEFAULT 'NO';
DECLARE
    CURSOR c_customers IS
        SELECT CustomerID, Balance
        FROM Customers;
    v_CustomerID Customers.CustomerID%TYPE;
    v_Balance NUMBER;
BEGIN
    FOR customer_rec IN c_customers LOOP
        v_CustomerID := customer_rec.CustomerID;
        v_Balance := customer_rec.Balance;
```

```
        IF v_Balance > 10000 THEN
            UPDATE Customers
            SET IsVIP = 'YES'
            WHERE CustomerID = v_CustomerID;
        ELSE
            UPDATE Customers
            SET IsVIP = 'NO'
            WHERE CustomerID = v_CustomerID;
        END IF;
    END LOOP;
    COMMIT;
END;
```

Scenario 3: The bank wants to send reminders to customers whose loans are due within the next 30 days.

Question: Write a PL/SQL block that fetches all loans due in the next 30 days and prints a reminder message for each customer.

```
SET SERVEROUTPUT ON;
DECLARE
    CURSOR c_loans IS
        SELECT LoanID, CustomerID, EndDate
        FROM Loans
        WHERE EndDate BETWEEN SYSDATE AND SYSDATE + 30;
    v_LoanID Loans.LoanID%TYPE;
    v_CustomerID Loans.CustomerID%TYPE;
    v_EndDate Loans.EndDate%TYPE;
    v_Name Customers.Name%TYPE;
BEGIN
    FOR loan_rec IN c_loans LOOP
        v_LoanID := loan_rec.LoanID;
```

```
        v_CustomerID := loan_rec.CustomerID;
        v_EndDate := loan_rec.EndDate;



        SELECT Name INTO v_Name
        FROM Customers
        WHERE CustomerID = v_CustomerID;



        DBMS_OUTPUT.PUT_LINE('Reminder: Customer ' || v_Name || ' (ID: ' ||
v_CustomerID ||
                             ') has a loan (ID: ' || v_LoanID || ') due on ' || TO_CHAR(v_EndDate,
'YYYY-MM-DD') || '.');
    END LOOP;
END;
/
```

Exercise 2: Error Handling

Scenario 1: Handle exceptions during fund transfers between accounts.

- **Question:** Write a stored procedure **SafeTransferFunds** that transfers funds between two accounts. Ensure that if any error occurs (e.g., insufficient funds), an appropriate error message is logged and the transaction is rolled back.

```
CREATE OR REPLACE PROCEDURE SafeTransferFunds(
    p_SourceAccountID IN Accounts.AccountID%TYPE,
    p_TargetAccountID IN Accounts.AccountID%TYPE,
    p_Amount IN Accounts.Balance%TYPE
) IS
    insufficient_funds EXCEPTION;
    v_SourceBalance Accounts.Balance%TYPE;
```

```
    v_TargetBalance Accounts.Balance%TYPE;
BEGIN
    -- Get the current balance of the source account
    SELECT Balance INTO v_SourceBalance
    FROM Accounts
    WHERE AccountID = p_SourceAccountID FOR UPDATE;

    -- Check if the source account has sufficient funds
    IF v_SourceBalance < p_Amount THEN
        RAISE insufficient_funds;
    END IF;

    -- Deduct the amount from the source account
    UPDATE Accounts
    SET Balance = Balance - p_Amount
    WHERE AccountID = p_SourceAccountID;

    -- Add the amount to the target account
    UPDATE Accounts
    SET Balance = Balance + p_Amount
    WHERE AccountID = p_TargetAccountID;

    COMMIT;

    EXCEPTION
        WHEN insufficient_funds THEN
            DBMS_OUTPUT.PUT_LINE('Error: Insufficient funds in source account.');
            ROLLBACK;
        WHEN OTHERS THEN
```

```
        DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
        ROLLBACK;
END SafeTransferFunds;
/
```

Scenario 2: Manage errors when updating employee salaries.

- **Question:** Write a stored procedure **UpdateSalary** that increases the salary of an employee by a given percentage. If the employee ID does not exist, handle the exception and log an error message.

```
CREATE OR REPLACE PROCEDURE UpdateSalary(
    p_EmployeeID IN Employees.EmployeeID%TYPE,
    p_Percentage IN NUMBER
) IS
    employee_not_found EXCEPTION;
    v_Salary Employees.Salary%TYPE;
BEGIN
    -- Check if employee exists
    SELECT Salary INTO v_Salary
    FROM Employees
    WHERE EmployeeID = p_EmployeeID FOR UPDATE;

    -- Update the salary
    UPDATE Employees
    SET Salary = Salary * (1 + p_Percentage / 100)
    WHERE EmployeeID = p_EmployeeID;

    COMMIT;

    EXCEPTION
        WHEN NO_DATA_FOUND THEN
```

```
        DBMS_OUTPUT.PUT_LINE('Error: Employee ID not found.');
        RAISE employee_not_found;
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
        ROLLBACK;
END UpdateSalary;
/
```

Scenario 3: Ensure data integrity when adding a new customer.

- **Question:** Write a stored procedure **AddNewCustomer** that inserts a new customer into the Customers table. If a customer with the same ID already exists, handle the exception by logging an error and preventing the insertion.

```
CREATE OR REPLACE PROCEDURE AddNewCustomer(
    p_CustomerID IN Customers.CustomerID%TYPE,
    p_Name IN Customers.Name%TYPE,
    p_DOB IN Customers.DOB%TYPE,
    p_Balance IN Customers.Balance%TYPE
) IS
    customer_already_exists EXCEPTION;
BEGIN
    -- Check if customer already exists
    SELECT CustomerID INTO p_CustomerID
    FROM Customers
    WHERE CustomerID = p_CustomerID;

    -- If the customer exists, raise an exception
    RAISE customer_already_exists;

    EXCEPTION
        WHEN NO_DATA_FOUND THEN
```

```
        -- Insert the new customer
        INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
        VALUES (p_CustomerID, p_Name, p_DOB, p_Balance, SYSDATE);
        COMMIT;
    WHEN customer_already_exists THEN
        DBMS_OUTPUT.PUT_LINE('Error: Customer with the same ID already
exists.');
        ROLLBACK;
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
        ROLLBACK;
END AddNewCustomer;
/
```

Exercise 3: Stored Procedures

Scenario 1: The bank needs to process monthly interest for all savings accounts.

- **Question:** Write a stored procedure **ProcessMonthlyInterest** that calculates and updates the balance of all savings accounts by applying an interest rate of 1% to the current balance.

```
CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS
BEGIN
  UPDATE Accounts
  SET Balance = Balance * 1.01
  WHERE AccountType = 'Savings';
  COMMIT;
END ProcessMonthlyInterest;
/
```

Scenario 2: The bank wants to implement a bonus scheme for employees based on their performance.

- **Question:** Write a stored procedure **UpdateEmployeeBonus** that updates the salary of employees in a given department by adding a bonus percentage passed as a parameter.

```
CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(
    p_Department IN Employees.Department%TYPE,
    p_BonusPercentage IN NUMBER
) IS
BEGIN
    UPDATE Employees
    SET Salary = Salary * (1 + p_BonusPercentage / 100)
    WHERE Department = p_Department;
    COMMIT;
END UpdateEmployeeBonus;
/
```

Scenario 3: Customers should be able to transfer funds between their accounts.

- **Question:** Write a stored procedure **TransferFunds** that transfers a specified amount from one account to another, checking that the source account has sufficient balance before making the transfer.

```
CREATE OR REPLACE PROCEDURE TransferFunds(
    p_SourceAccountID IN Accounts.AccountID%TYPE,
    p_TargetAccountID IN Accounts.AccountID%TYPE,
    p_Amount IN Accounts.Balance%TYPE
) IS
    insufficient_funds EXCEPTION;
    v_SourceBalance Accounts.Balance%TYPE;
    v_TargetBalance Accounts.Balance%TYPE;
BEGIN
    -- Get the current balance of the source account
```

```
    SELECT Balance INTO v_SourceBalance
    FROM Accounts
    WHERE AccountID = p_SourceAccountID FOR UPDATE;

    -- Check if the source account has sufficient funds
    IF v_SourceBalance < p_Amount THEN
        RAISE insufficient_funds;
    END IF;

    -- Deduct the amount from the source account
    UPDATE Accounts
    SET Balance = Balance - p_Amount
    WHERE AccountID = p_SourceAccountID;

    -- Add the amount to the target account
    UPDATE Accounts
    SET Balance = Balance + p_Amount
    WHERE AccountID = p_TargetAccountID;

    COMMIT;

    EXCEPTION
        WHEN insufficient_funds THEN
            DBMS_OUTPUT.PUT_LINE('Error: Insufficient funds in source account.');
            ROLLBACK;
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
            ROLLBACK;
END TransferFunds;
```

/

Exercise 4: Functions

Scenario 1: Calculate the age of customers for eligibility checks.

- **Question:** Write a function CalculateAge that takes a customer's date of birth as input and returns their age in years.

```
CREATE OR REPLACE FUNCTION CalculateAge(
    p_DOB IN DATE
) RETURN NUMBER IS
    v_Age NUMBER;
BEGIN
    v_Age := FLOOR(MONTHS_BETWEEN(SYSDATE, p_DOB) / 12);
    RETURN v_Age;
END CalculateAge;
/
```

Scenario 2: The bank needs to compute the monthly installment for a loan.

- **Question:** Write a function **CalculateMonthlyInstallment** that takes the loan amount, interest rate, and loan duration in years as input and returns the monthly installment amount.

```
CREATE OR REPLACE FUNCTION CalculateMonthlyInstallment(
    p_LoanAmount IN NUMBER,
    p_InterestRate IN NUMBER,
    p_LoanDurationYears IN NUMBER
) RETURN NUMBER IS
    v_MonthlyInstallment NUMBER;
    v_MonthlyRate NUMBER := p_InterestRate / 12 / 100;
    v_TotalMonths NUMBER := p_LoanDurationYears * 12;
BEGIN
```

```
    v_MonthlyInstallment := p_LoanAmount * v_MonthlyRate / (1 - POWER(1 +
v_MonthlyRate, -v_TotalMonths));
    RETURN v_MonthlyInstallment;
END CalculateMonthlyInstallment;
/
```

Scenario 3: Check if a customer has sufficient balance before making a transaction.

- **Question:** Write a function **HasSufficientBalance** that takes an account ID and an amount as input and returns a boolean indicating whether the account has at least the specified amount.

```
CREATE OR REPLACE FUNCTION HasSufficientBalance(
    p_AccountID IN Accounts.AccountID%TYPE,
    p_Amount IN NUMBER
) RETURN BOOLEAN IS
    v_Balance Accounts.Balance%TYPE;
BEGIN
    SELECT Balance INTO v_Balance
    FROM Accounts
    WHERE AccountID = p_AccountID;

    IF v_Balance >= p_Amount THEN
        RETURN TRUE;
    ELSE
        RETURN FALSE;
    END IF;
END HasSufficientBalance;
/
```

Exercise 5: Triggers

Scenario 1: Automatically update the last modified date when a customer's record is updated.

- **Question:** Write a trigger **UpdateCustomerLastModified** that updates the LastModified column of the Customers table to the current date whenever a customer's record is updated.

```
CREATE OR REPLACE TRIGGER UpdateCustomerLastModified
BEFORE UPDATE ON Customers
FOR EACH ROW
BEGIN
   :NEW.LastModified := SYSDATE;
END UpdateCustomerLastModified;
/
```

Scenario 2: Maintain an audit log for all transactions.

- **Question:** Write a trigger **LogTransaction** that inserts a record into an AuditLog table whenever a transaction is inserted into the Transactions table.

```
CREATE TABLE AuditLog (
   LogID NUMBER PRIMARY KEY,
   TransactionID NUMBER,
   AccountID NUMBER,
   TransactionDate DATE,
   Amount NUMBER,
   TransactionType VARCHAR2(10),
   LogTimestamp DATE
);


CREATE OR REPLACE TRIGGER LogTransaction
AFTER INSERT ON Transactions
FOR EACH ROW
BEGIN
   INSERT INTO AuditLog (LogID, TransactionID, AccountID, TransactionDate, Amount, TransactionType, LogTimestamp)
```

```
    VALUES (AuditLog_SEQ.NEXTVAL, :NEW.TransactionID, :NEW.AccountID,
:NEW.TransactionDate, :NEW.Amount, :NEW.TransactionType, SYSDATE);
END LogTransaction;
/
```

Scenario 3: Enforce business rules on deposits and withdrawals.

- **Question:** Write a trigger **CheckTransactionRules** that ensures withdrawals do not exceed the balance and deposits are positive before inserting a record into the Transactions table.

```
CREATE OR REPLACE TRIGGER CheckTransactionRules
BEFORE INSERT ON Transactions
FOR EACH ROW
DECLARE
    v_Balance Accounts.Balance%TYPE;
BEGIN
    -- Get the current balance of the account
    SELECT Balance INTO v_Balance
    FROM Accounts
    WHERE AccountID = :NEW.AccountID FOR UPDATE;

    -- Check if the transaction is a withdrawal
    IF :NEW.TransactionType = 'Withdrawal' THEN
        IF v_Balance < :NEW.Amount THEN
            RAISE_APPLICATION_ERROR(-20001, 'Error: Insufficient balance for
withdrawal.');
        END IF;
    ELSIF :NEW.TransactionType = 'Deposit' THEN
        IF :NEW.Amount <= 0 THEN
            RAISE_APPLICATION_ERROR(-20002, 'Error: Deposit amount must be
positive.');
        END IF;
```

```
    ELSE
        RAISE_APPLICATION_ERROR(-20003, 'Error: Invalid transaction type.');
    END IF;
END CheckTransactionRules;
/
```

Exercise 6: Cursors

Scenario 1: Generate monthly statements for all customers.

- **Question:** Write a PL/SQL block using an explicit cursor **GenerateMonthlyStatements** that retrieves all transactions for the current month and prints a statement for each customer.

```
DECLARE
    CURSOR c_transactions IS
        SELECT CustomerID, AccountID, TransactionDate, Amount, TransactionType
        FROM Transactions
        WHERE TransactionDate BETWEEN TRUNC(SYSDATE, 'MM') AND
LAST_DAY(SYSDATE);
    v_CustomerID Transactions.CustomerID%TYPE;
    v_AccountID Transactions.AccountID%TYPE;
    v_TransactionDate Transactions.TransactionDate%TYPE;
    v_Amount Transactions.Amount%TYPE;
    v_TransactionType Transactions.TransactionType%TYPE;
BEGIN
    OPEN c_transactions;
    LOOP
        FETCH c_transactions INTO v_CustomerID, v_AccountID, v_TransactionDate,
v_Amount, v_TransactionType;
        EXIT WHEN c_transactions%NOTFOUND;

        -- Print statement for each transaction
```

```
        DBMS_OUTPUT.PUT_LINE('Customer ID: ' || v_CustomerID || ', Account ID: '
|| v_AccountID ||
                             ', Transaction Date: ' || TO_CHAR(v_TransactionDate, 'YYYY-MM-
DD') ||
                             ', Amount: ' || v_Amount || ', Transaction Type: ' ||
v_TransactionType);
    END LOOP;
    CLOSE c_transactions;
END;
```

Scenario 2: Apply annual fee to all accounts.

- **Question:** Write a PL/SQL block using an explicit cursor **ApplyAnnualFee** that deducts an annual maintenance fee from the balance of all accounts.

```
DECLARE
    CURSOR c_accounts IS
        SELECT AccountID, Balance
        FROM Accounts;
    v_AccountID Accounts.AccountID%TYPE;
    v_Balance Accounts.Balance%TYPE;
    v_AnnualFee CONSTANT NUMBER := 50; -- Example annual fee
BEGIN
    OPEN c_accounts;
    LOOP
        FETCH c_accounts INTO v_AccountID, v_Balance;
        EXIT WHEN c_accounts%NOTFOUND;

        -- Deduct annual fee from each account
        UPDATE Accounts
        SET Balance = Balance - v_AnnualFee
        WHERE AccountID = v_AccountID;
    END LOOP;
```

```
    CLOSE c_accounts;
    COMMIT;
END;
/
```

Scenario 3: Update the interest rate for all loans based on a new policy.

- **Question:** Write a PL/SQL block using an explicit cursor **UpdateLoanInterestRates** that fetches all loans and updates their interest rates based on the new policy.

```
DECLARE
    CURSOR c_loans IS
        SELECT LoanID, InterestRate
        FROM Loans;
    v_LoanID Loans.LoanID%TYPE;
    v_InterestRate Loans.InterestRate%TYPE;
    v_NewInterestRate CONSTANT NUMBER := 3.5; -- Example new interest rate
BEGIN
    OPEN c_loans;
    LOOP
        FETCH c_loans INTO v_LoanID, v_InterestRate;
        EXIT WHEN c_loans%NOTFOUND;

        -- Update the interest rate for each loan
        UPDATE Loans
        SET InterestRate = v_NewInterestRate
        WHERE LoanID = v_LoanID;
    END LOOP;
    CLOSE c_loans;
    COMMIT;
END;
/
```

Exercise 7: Packages

Scenario 1: Group all customer-related procedures and functions into a package.

- **Question:** Create a package **CustomerManagement** with procedures for adding a new customer, updating customer details, and a function to get customer balance.

```
CREATE OR REPLACE PACKAGE CustomerManagement IS
    PROCEDURE AddCustomer(
        p_CustomerID IN Customers.CustomerID%TYPE,
        p_Name IN Customers.Name%TYPE,
        p_DOB IN Customers.DOB%TYPE,
        p_Balance IN Customers.Balance%TYPE);

    PROCEDURE UpdateCustomer(
        p_CustomerID IN Customers.CustomerID%TYPE,
        p_Name IN Customers.Name%TYPE,
        p_DOB IN Customers.DOB%TYPE,
        p_Balance IN Customers.Balance%TYPE);

    FUNCTION GetCustomerBalance(
        p_CustomerID IN Customers.CustomerID%TYPE) RETURN NUMBER;
END CustomerManagement;
/
```

Scenario 2: Create a package to manage employee data.

- **Question:** Write a package **EmployeeManagement** with procedures to hire new employees, update employee details, and a function to calculate annual salary.

```
CREATE OR REPLACE PACKAGE BODY CustomerManagement IS
    PROCEDURE AddCustomer(
        p_CustomerID IN Customers.CustomerID%TYPE,
        p_Name IN Customers.Name%TYPE,
```

```
    p_DOB IN Customers.DOB%TYPE,
    p_Balance IN Customers.Balance%TYPE) IS
  BEGIN
    INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)
    VALUES (p_CustomerID, p_Name, p_DOB, p_Balance, SYSDATE);
    COMMIT;
  END AddCustomer;

  PROCEDURE UpdateCustomer(
    p_CustomerID IN Customers.CustomerID%TYPE,
    p_Name IN Customers.Name%TYPE,
    p_DOB IN Customers.DOB%TYPE,
    p_Balance IN Customers.Balance%TYPE) IS
  BEGIN
    UPDATE Customers
    SET Name = p_Name, DOB = p_DOB, Balance = p_Balance, LastModified =
SYSDATE
    WHERE CustomerID = p_CustomerID;
    COMMIT;
  END UpdateCustomer;

  FUNCTION GetCustomerBalance(
    p_CustomerID IN Customers.CustomerID%TYPE) RETURN NUMBER IS
    v_Balance Customers.Balance%TYPE;
  BEGIN
    SELECT Balance INTO v_Balance
    FROM Customers
    WHERE CustomerID = p_CustomerID;
    RETURN v_Balance;
```

```
    END GetCustomerBalance;
END CustomerManagement;
/
```

Scenario 3: Group all account-related operations into a package.

- **Question:** Create a package **AccountOperations** with procedures for opening a new account, closing an account, and a function to get the total balance of a customer across all accounts.

```
CREATE OR REPLACE PACKAGE EmployeeManagement IS
    PROCEDURE HireEmployee(
        p_EmployeeID IN Employees.EmployeeID%TYPE,
        p_Name IN Employees.Name%TYPE,
        p_Position IN Employees.Position%TYPE,
        p_Salary IN Employees.Salary%TYPE,
        p_Department IN Employees.Department%TYPE,
        p_HireDate IN Employees.HireDate%TYPE);

    PROCEDURE UpdateEmployeeDetails(
        p_EmployeeID IN Employees.EmployeeID%TYPE,
        p_Name IN Employees.Name%TYPE,
        p_Position IN Employees.Position%TYPE,
        p_Salary IN Employees.Salary%TYPE,
        p_Department IN Employees.Department%TYPE);

    FUNCTION CalculateAnnualSalary(
        p_EmployeeID IN Employees.EmployeeID%TYPE) RETURN NUMBER;
END EmployeeManagement;
/
```